Restricted Dynamic Programming Heuristic for Precedence Constrained Bottleneck Generalized TSP

Yaroslav Salii[1,2]

[1] IMM UB RAS, Yekaterinburg, Russia,
[2] Ural Federal University, Yekaterinburg, Russia
yvs314@gmail.com

Abstract. We develop a restricted dynamical programming heuristic for a complicated traveling salesman problem: a)cities are grouped into clusters, resp. Generalized TSP; b)precedence constraints are imposed on the order of visiting the clusters, resp. Precedence Constrained TSP; c)the costs of moving to the next cluster and doing the required job inside one are aggregated in a minimax manner, resp. Bottleneck TSP; d)all the costs may depend on the sequence of previously visited clusters, resp. Sequence-Dependent TSP or Time Dependent TSP. Such multiplicity of constraints complicates the use of mixed integer-linear programming, while dynamic programming (DP) benefits from them; the latter may be supplemented with a branch-and-bound strategy, which necessitates a "DP-compliant" heuristic. The proposed heuristic always yields a feasible solution, which is not always the case with heuristics, and its precision may be tuned until it becomes the exact DP.

Keywords: sequential ordering problem · traveling salesman · dynamic programming · precedence constraints · generalized traveling salesman · bottleneck traveling salesman

Introduction

Our object is a particularly complicated version of the well-known Traveling Salesman Problem (TSP), which combines several its generalizations that are usually treated separately: Bottleneck TSP [23, Ch. 15], Generalized TSP [23, Ch. 12], Precedence Constrained TSP [40, 18, 29, 8], Sequence-Dependent[3] TSP [3] (as a generalization of the more well-known Time-Dependent TSP [20]). Nevertheless, their combination is neither a purely scholastic effort nor art for art's

[3] Note that the *Sequence Dependent* designation is mostly applied to scheduling problems, see [4], and has a different meaning: the cost of *present* action only depends on the cost of the *previous* one, not on the entire previous sequence. To the best of the author's knowledge, the only term for dependence on the whole (previous) sequence was *State Dependent*, introduced in [35] for a scheduling problem concerning printed circuit board assembly; nevertheless, we prefer to follow to the 'Sequence Dependent' designation of [3].

sake. Its possible applications include printed circuit boards design (a less general version, without precedence constraints, was considered in [25]) and optimization of the cycle time for industrial robots (for a survey of robotic task sequencing, refer to [2]). See the relevant reviews of TSP and its variations in [34, 37, 31, 23, 30]. For a most recent treatment of Precedence Constrained TSP (TSP-PC), also known as Sequential Ordering Problem (SOP), see [19].

A Terminological note We call the variation of TSP we consider in this paper a Sequence-Dependent Precedence Constrained Bottleneck Generalized TSP (SD-BGTSP-PC); however, since the sequence dependence does not currently play an important part in our model, we will mostly omit that designation and refer to our problem as BGTSP-PC, bearing the possible sequence dependence in mind. The designation TSP-PC for Precedence Constrained TSP is borrowed from [8]; we find it appealing, since it poses no risk of confusion with the very different Prize Collecting TSP, which is also abbreviated PCTSP [23, Ch. 14], makes an explicit reference to the "ordinary" TSP (in contrast with SOP), and does not specifically mention asymmetricity (our method is symmetricity-agnostic), in contrast with PCATS [6]. Another important issue is that we consider an *open* problem of the TSP family, i.e., return to origin is not mandated; to the best of our knowledge, the open TSP was first posed in [15], with a reference to a 1965 report by N.Deo and S.L.Hakimi, as "Shortest Hamiltonian Chain Problem".

The rest of the paper is as follows: in Sect. 1 we describe general notation and definitions; Sect. 2 describes the problem statement, and Sect. 3 completes the description with definitions of dynamic programming subproblems and the Bellman Equation. Section 4 describes the exact dynamic programming for our problem and Sect. 5 describes our experience of shared memory parallelization of this algorithm. Section 6 discusses the proposed heuristic, reports and compares the results of experiments with the parallel implementation of the exact algorithm and the proposed heuristic. Sections 1–4 follow the most recent paper on exact solution of the problem [13],

1 General notation and definitions

We employ the standard set-theoretic notation (quantifiers, propositional connectives, etc.); $\triangleq$ denotes *equality by definition*. Each set, all elements of which are sets themselves, is called a *family*. For every two objects a and b, denote by $\{a; b\}$ the (unique) set that contains a, b, and nothing else. In the case $a = b$, this yields a *singleton* $\{a\} = \{b\}$. We employ the standard Kuratovskii ordered pair definition: for two arbitrary objects u and v, their *ordered pair* (OP) is defined by $(u, v) \triangleq \{\{u\}; \{u; v\}\}$; its first element is u and the second one is v. For an OP z, $\mathrm{pr}_1(z)$ denotes its first element and $\mathrm{pr}_2(z)$ denotes its second element; these are uniquely defined by the condition $z = (\mathrm{pr}_1(z), \mathrm{pr}_2(z))$; in case $z \in A \times B$, where A and B are sets, we have $\mathrm{pr}_1(z) \in A$ and $\mathrm{pr}_2(z) \in B$. We employ the usual canonical representation of ordered *triplet* [16, § 1.3]: for three objects a, b, and c, we assume $(a, b, c) \triangleq ((a, b), c)$. A similar convention is used for Cartesian product

of three sets: for arbitrary sets A, B, and C, we have $A \times B \times C \triangleq (A \times B) \times C$ [16, § 1.3]; it obviously means that $(x, y) \in A \times B \times C \,\forall x \in A \times B \,\forall y \in C$. In connection with this, let us also recall the convention concerning the notation for values of function of three variables: for sets A, B, C, and D, function $h : A \times B \times C \to D$ and elements $\mu \in A \times B$ and $\nu \in C$, in accordance with the above-mentioned representation of $A \times B \times C$, it is valid to consider the element $h(\mu, \nu) \in D$ to be defined.

As usual, $[0, \infty[\triangleq \{\xi \in \mathbb{R} | 0 \leq \xi\}$ ($\mathbb{R}$ is the real line). For each nonempty set S, denote by $\mathcal{R}_+[S]$ the set of all (nonnegative) functions from S to $[0, \infty[$. As usual, $\mathbb{N} \triangleq \{1; 2; \ldots\}$. Assume $\mathbb{N}_0 \triangleq \{0\} \cup \mathbb{N}$ and $\overline{p, q} \triangleq \{i \in \mathbb{N}_0 | (p \leq i) \wedge (i \leq q)\}$ $\forall p \in \mathbb{N}_0 \,\forall q \in \mathbb{N}_0$. Note that the latter definition yields the *empty set* if $p > q$.

For a nonempty finite set K, let $|K| \in \mathbb{N}$ be the *power* of the set K; then, let (bi)$[K]$ denote the set of *all bijections* of the "interval" $\overline{1, |K|}$ onto K. In particular, for a fixed $N \in \mathbb{N}$, let $\mathbb{P} \triangleq$ (bi)$[\overline{1, N}]$ be the set of all *permutations* of the "interval" $\overline{1, N}$; for each $\lambda \in \mathbb{P}$, there exists a permutation $\lambda^{-1} \in \mathbb{P}$ such that $\lambda(\lambda^{-1}(k)) = \lambda^{-1}(\lambda(k)) = k$ $\forall k \in \overline{1, N}$. Denote by $\mathcal{P}(H)$ ($\mathcal{P}'(H)$) the family of all (all nonempty) subsets of set H; let $\mathrm{Fin}(H)$ be the family of all finite sets from $\mathcal{P}'(H)$.

2 Problem statement

Here and below, fix a nonempty set X, where everything happens, a point $x^0 \in X$, which is called the *base*, a natural number N, $N \geq 2$, which is the main dimension parameter, sets

$$M_1 \in \mathrm{Fin}\, X, \ldots, M_N \in \mathrm{Fin}\, X,$$

referred to as *megalopolises*, and relations

$$\mathbb{M}_1 \in \mathcal{P}'(M_1 \times M_1), \ldots, \mathbb{M}_N \in \mathcal{P}'(M_N \times M_N). \tag{1}$$

For $j \in \overline{1, N}$, OPs $z \in \mathbb{M}_j$ describe the possible ways of conducting interior jobs inside the megalopolis M_j: $\mathrm{pr}_1(z)$ determines the *entry point* and $\mathrm{pr}_2(z)$ determines the *exit point*. The scheme of movements is as follows:

$$\begin{aligned} \left(x^0\right) \to \left(\mathrm{pr}_1\left(z^{(1)}\right) \in M_{\alpha(1)} \rightsquigarrow \mathrm{pr}_2\left(z^{(1)}\right) \in M_{\alpha(1)}\right) \to \\ \to \left(\mathrm{pr}_1\left(z^{(2)}\right) \in M_{\alpha(2)} \rightsquigarrow \mathrm{pr}_2\left(z^{(2)}\right) \in M_{\alpha(2)}\right) \to \\ \to \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad \to \\ \to \left(\mathrm{pr}_1\left(z^{(N)}\right) \in M_{\alpha(N)} \rightsquigarrow \mathrm{pr}_2\left(z^{(N)}\right) \in M_{\alpha(N)}\right). \end{aligned} \tag{2}$$

where α is a permutation of indices from $\overline{1, N}$ and OPs $z^{(1)}, \ldots, z^{(N)}$ satisfy the conditions

$$z^{(1)} \in \mathbb{M}_{\alpha(1)}, \ldots, z^{(N)} \in \mathbb{M}_{\alpha(N)}. \tag{3}$$

In (2), we choose the permutation α, in our terms, the *route*, and a tuple $(z^{(1)}, \ldots, z^{(N)})$ that agrees with the route in the sense of (3); this tuple is called a *track*. Let us stress that the choice of α may be restricted by *precedence constraints*, which will be introduced below. Megalopolises are assumed to be disjoint, and the base does not belong to any one of them:

$$(x^0 \notin M_j \ \forall j \in \overline{1,N}) \wedge (M_p \cap M_q = \varnothing \ \forall p \in \overline{1,N} \ \forall q \in \overline{1,N} \setminus \{p\}). \tag{4}$$

Although this convention is rather common, there are engineering problems where it does not hold and the megalopolises could intersect [17]. For greater clarity in definitions below, let us gather the possible entry points[4] and exit points into separate sets for each megalopolis:

$$\begin{aligned} \mathbb{M}_j^{(\mathrm{in})} &\triangleq \{\mathrm{pr}_1(z) : z \in \mathbb{M}_j\} \ \forall j \in \overline{1,N}, \\ \mathbb{M}_j^{(\mathrm{out})} &\triangleq \{\mathrm{pr}_2(z) : z \in \mathbb{M}_j\} \ \forall j \in \overline{1,N}. \end{aligned} \tag{5}$$

In terms of the megalopolises and sets (5), let us describe three specific nonempty subsets of X:

$$\mathbb{X} \triangleq \{x^0\} \cup \left(\bigcup_{i=1}^{N} M_i \right), \tag{6}$$

$$\mathbb{X}_{\mathrm{in}} \triangleq \{x^0\} \cup \left(\bigcup_{i=1}^{N} \mathbb{M}_i^{(\mathrm{in})} \right) \cup \left(\{\varnothing\} \right), \tag{7}$$

$$\mathbb{X}_{\mathrm{out}} \triangleq \{x^0\} \cup \left(\bigcup_{i=1}^{N} \mathbb{M}_i^{(\mathrm{out})} \right); \tag{8}$$

clearly, $\mathbb{X}_{\mathrm{in}} \subset \mathbb{X}$ $\mathbb{X}_{\mathrm{out}} \subset \mathbb{X}$. Like (5), these three sets serve to clarify the future definitions, a kind of syntactic sugar. The "empty" set $\{\varnothing\}$ in $\mathbb{X}_{\mathrm{in}}$ has a special meaning: it signifies that the entry point is irrelevant. It is worth mention that generally $\mathbb{X} \neq X$: we often deal with Euclidean plane $X = \mathbb{R}^2$, whereas the problem itself revolves around the discrete set of points $\mathbb{X}$; the case of continuous M_i is also worth mention, it is known as Generalized TSP with Neighborhoods (GTSPN), see [2, 28]. It has also been studied since the end of 1980s by L.N. Korotayeva and A.G. Chentsov and their colleagues [26, 25] under the plain label of GTSP or "Routing Problem".

The author is aware of the three means to define precedence constraints:

- Partial order [46];
- Directed acyclic graph [18];
- Nondescript binary relation (a set of OPs) [11].

Clearly, these approaches produce the same results (all may be reduced to a kind of binary relation, note also the result of [1] on path information) and their

[4] the terms "city" and "point" are used interchangeably

choice is mostly a matter of taste and specific objectives of a paper. We find the third approach to be the most convenient means of expressing the precedence constraints for a dynamic programming procedure. Let us introduce the set $\mathbb{K} \in \mathcal{P}(\overline{1,N} \times \overline{1,N})$ of OPs and call its elements *address pairs*. In an address pair $h \in \mathbb{K}$, the first element $\mathrm{pr}_1(h) \in \overline{1,N}$ is called a *sender*, and the second one $\mathrm{pr}_2(h) \in \overline{1,N}$ a *receiver*. The essence of precedence constraints is that for each pair the sender *must* be visited before the receiver. The case $\mathbb{K} = \varnothing$ is not excluded and corresponds to the lack of precedence constraints, although in this case it is probably better to forgo Dynamic Programming and implement a more usual branch-and-cut algorithm (see the description in [5]).

Recall that $\mathbb{P} = (\mathrm{bi})[\overline{1,N}]$ is the set of all (complete) routes; it is a nonempty set of cardinality $|P| = N!$. In terms of address pairs, the set of feasible routes is expressed as follows (see [11, Pt. 2]):

$$\mathbb{A} \triangleq \left\{ \alpha \in \mathbb{P} \,\middle|\, \alpha^{-1}\big(\mathrm{pr}_1(h)\big) < \alpha^{-1}\big(\mathrm{pr}_2(h)\big) \;\forall h \in \mathbb{K} \right\}. \tag{9}$$

Since we use generic, nondescript binary relation, we must impose the following condition to ensure the existence of feasible routes (see [11, Pt. 2]):

$$\forall\, \mathbb{K}_0 \in \mathcal{P}'(\mathbb{K})\; \exists z_0 \in \mathbb{K}_0 : \mathrm{pr}_1(z_0) \neq \mathrm{pr}_2(z)\; \forall z \in \mathbb{K}_0; \tag{10}$$

it implies that, in particular, $\mathrm{pr}_1(z) \neq \mathrm{pr}_2(z)\; \forall z \in \mathbb{K}$ and is clearly equivalent to the condition of *acyclicity* for the corresponding precedence digraph. One may characterize $\mathbb{A}$ as the set of all routes $\alpha \in \mathbb{P}$ such that

$$\Big(\big(\mathrm{pr}_1(z) = \alpha(t_1)\big) \wedge \big(\mathrm{pr}_2(z) = \alpha(t_2)\big)\Big) \Rightarrow (t_1 < t_2)$$

for an address pair $z \in \mathbb{K}$ and "times" $t_1 \in \overline{1,N}$ and $t_2 \in \overline{1,N}$. In addition to the route, we also choose the *track*, or trajectory, which is determined (2) by the OPs $z^{(1)}, \ldots, z^{(N)}$, supplemented with the initial OP $\big(\{\varnothing\}, x^0\big)$. To formally define the set of tracks that agree with some route in the sense of (2), denote by $\mathcal{Z}$ the set of all tuples $(z_i)_{i=0}^N : \overline{0,N} \to \mathbb{X}_{\mathrm{in}} \times \mathbb{X}_{\mathrm{out}}$. For $\alpha \in \mathbb{P}$, assume

$$\mathcal{Z}^{(\alpha)} \triangleq \left\{ (z_i)_{i=0}^N \in \mathcal{Z} \,\middle|\, \big(z_0 = (x^0, x^0)\big) \wedge \big(z_t \in \mathbb{M}_{\alpha(t)}\; \forall t \in \overline{1,N}\big) \right\}; \tag{11}$$

evidently, $\mathcal{Z}_\alpha \in \mathrm{Fin}(\mathcal{Z})$.

At last, we can proceed to the definition of the quality criterion for our problem. To account for the influence of the set of pending tasks on the cost function (recall that our problem is *sequence dependent*), we will need the symbol $\mathfrak{N} \triangleq \mathcal{P}'(\overline{1,N})$ (we call an element of $\mathfrak{N}$ a *task set*). Now, let us introduce the following $N+1$ *cost functions*

$$\mathbf{c} \in \mathcal{R}_+(\mathbb{X}_{\mathrm{out}} \times \mathbb{X}_{\mathrm{in}} \times \mathfrak{N}), c_1 \in \mathcal{R}_+(\mathbb{X}_{\mathrm{in}} \times \mathbb{X}_{\mathrm{out}} \times \mathfrak{N}), \ldots, c_N \in \mathcal{R}_+(\mathbb{X}_{\mathrm{in}} \times \mathbb{X}_{\mathrm{out}} \times \mathfrak{N}). \tag{12}$$

For $\alpha \in \mathbb{P}$ and $(z_i)_{i=0}^N \in \mathcal{Z}^{(\alpha)}$, assume

$$\mathfrak{C}^{(\alpha)}\left[(z_i)_{i=0}^N\right] \triangleq \max_{t\in\overline{0,N-1}} \Bigg[\mathbf{c}(\mathrm{pr}_2(z_t), \mathrm{pr}_1(z_{t+1}), \{\alpha(s) : s \in \overline{t+1,N}\}) + \\ + c_{\alpha(t+1)}(z_{t+1}, \{\alpha(s) : s \in \overline{t+1,N}\})\Bigg]. \tag{13}$$

This is a case of *minimax* (bottleneck) aggregation of the summary cost of moving from the current megalopolis to the next, where the *exterior movement* cost is described by $\mathbf{c} \in \mathcal{R}_+(\mathbb{X}_{\mathrm{out}} \times \mathbb{X}_{\mathrm{in}} \times \mathfrak{N})$, and the cost of conducting the *interior job* in the next megalopolis, $c_{t+1} \in \mathcal{R}_+(\mathbb{X}_{\mathrm{in}} \times \mathbb{X}_{\mathrm{out}} \times \mathfrak{N})$.

The *interior jobs* setting introduced in [10] provides a means for universal expression of what we are to do inside a megalopolis, thereby unifying the *Generalized* TSP [23, Ch. 13] and *Clustered* TSP [14] approaches:

Generalized *or International* TSP. Each cluster is to be visited exactly once, i.e., only one city is to be visited per cluster. To adapt our statement to these requirements, we set $\forall i \in \overline{1,N}\ \mathbb{M}_i \triangleq \big\{(b,b) : b \in M_i\big\}$, i.e., we mandate exit at the point of entry for every megalopolis, and set the rudimentary zero interior job costs: $\forall i \in \overline{1,N}\, \forall b \in \mathbb{M}_i \forall K \in \mathfrak{N}\ c_i(b,b,K) = 0$.

Clustered TSP. For each cluster, all of its cities must be visited contiguously before proceeding to the next cluster, i.e., we have an open TSP (Shortest Hamiltonian Chain Problem [15]) inside each cluster thus we can never exit a cluster at the city we used to enter it, hence $\forall i \in \overline{1,N}\ \mathbb{M}_i \triangleq \big\{(a,b) : (a,b \in M_i) \wedge (b \neq a)\big\}$, and the costs of interior jobs are set to the costs of the respective open TSP tours starting at the respective entry points.

The problem statement for BGTSP-PC is as follows:

$$\mathfrak{C}^{(\alpha)}\left[(z_i)_{i=0}^N\right] = \max_{t\in\overline{0,N-1}} \Bigg[\mathbf{c}(\mathrm{pr}_2(z_t), \mathrm{pr}_1(z_{t+1}), \{\alpha(s) : s \in \overline{t+1,N}\}) + \\ + c_{\alpha(t+1)}(z_{t+1}, \{\alpha(s) : s \in \overline{t+1,N}\})\Bigg];$$

$$\mathfrak{C}^{(\alpha)}\left[(z_i)_{i=0}^N\right] \to \min, \alpha \in \mathbb{A}, (z_i)_{i=0}^N \in \mathcal{Z}^{(\alpha)}. \tag{BGTSP-PC}$$

Denote the *value* (extremum) of the problem by V,

$$V \triangleq \min_{\alpha\in\mathbb{A}} \min_{(z_i)_{i=0}^N \in \mathcal{Z}^{(\alpha)}} \mathfrak{C}^{(\alpha)}\left[(z_i)_{i=0}^N\right] \in [0,\infty[; \tag{14}$$

This extremum is attained by a *feasible solution*, expressed as an OP formed by the route and the track $\big(\alpha, (z_i)_{i=0}^N\big), \alpha \in \mathbf{A}, (z_i)_{i=0}^N \in \mathcal{Z}^{(\alpha)}$. A feasible solution $\big(\alpha^0, (z_i^0)_{i=0}^N\big)$ is considered *optimal* if $\mathfrak{C}_{\alpha^0}\left[(z_i^0)_{i=0}^N\right] = V$; there may be multiple optimal solutions.

3 Subproblems and Bellman equation

A dynamic programming (DP henceforth) solution consists of embedding a problem into a family of similar problems and obtaining a relation between the extrema and solutions of less difficult problems of the family and the full problem; this relation is expressed through a Bellman (recurrence) equation (see the general description of the method in [38, Ch. 9]). Note that the method below is an example of *backwards* DP akin to [7], whereas the *forward* DP [24, §1.2] seems to be more common as applied to the problems of the TSP family.

Precedence constraints pose a challenge when we attempt to reduce the full problem to a series of subproblems since they are formulated with respect to a *complete* set of megalopolises (in non-generalized TSP, cities) $\overline{1,N}$, and it is not immediately clear how to 'extend' precedence constraints to its subsets. The idea is to regard the set of megalopolises, which is to be visited in some subproblem, as a 'prefix' (the forward approach of [24, 33, 46, 8]) of some *feasible* route, or a 'suffix' (the backwards approach of [7, 11]) thereof. Naturally, such prefix or suffix has to be ordered consistently with the precedence constraints on a full route. Since we chose the *backwards* approach, in the following definitions we default to *suffixes*, with a possible passing reference to *prefixes*.

To properly describe the 'feasible suffixes', we need several new definitions. First of all, a subset K of the complete task set $\overline{1,N}$ is considered *feasible* if $\forall z \in \mathbb{K}\ (\mathrm{pr}_1(z) \in K) \Rightarrow (\mathrm{pr}_2(z) \in K)$. For prefixes, the implication is reversed, see [24, §1.3]. Any task set encountered below is to be assumed feasible unless explicitly stated otherwise, i.e., let us *redefine* $\mathfrak{N}$ to be not the set of all task sets but the set of all *feasible* task sets.

Infeasible task sets do not influence the solution of the problem, and it is this lack of influence that makes it possible to apply DP to precedence constrained problems of the TSP family thanks to a reduction of state space. For a quantification of this lack of influence and the respective reduction of state space, refer to [46, 47, 43].

For an arbitrary task set $K \in \mathfrak{N}$, consider the set $\Sigma[K]$ of the address pairs that are "saturated" in K:

$$\Sigma[K] \triangleq \left\{ z \in \mathbb{K} \;\middle|\; \big(\mathrm{pr}_1(z) \in K\big) \& \big(\mathrm{pr}_2(z) \in K\big) \right\}.$$

With its help, define the mapping $\mathbb{I} : \mathfrak{N} \to \mathfrak{N}$ as follows:

$$\mathbb{I}(\widetilde{K}) \triangleq \widetilde{K} \setminus \left\{ \mathrm{pr}_2(z) : z \in \Sigma[\widetilde{K}] \right\} \ \forall \widetilde{K} \in \mathfrak{N}. \tag{15}$$

We call the mapping $\mathbb{I}$ a *crossing-out rule*. For a task set $\widetilde{K}$, it specifies the possible "entry points", from which it is possible to start a walk through $\widetilde{K}$. From the partially ordered set perspective, the mapping $\mathbb{I}$ produces a specific maximum *antichain*, retaining, for each chain present in $\widetilde{K}$, only its *minimum* element. In a forward procedure, the respective mapping would retain the *maximum* element, see [46].

We can now define the set of *partial routes* through a task set $K \in \mathfrak{N}$, which we described above as 'feasible suffixes':

$$(\mathbb{I}-\mathrm{bi})[K] \triangleq \left\{ \alpha \in (\mathrm{bi})[K] \,\middle|\, \alpha(s) \in \mathbb{I}\big(\{\alpha(t) : t \in \overline{s,|K|}\}\big)\ \forall s \in \overline{1,|K|} \right\}. \quad (16)$$

This set is non-empty for feasible $K \in \mathfrak{N}$ (see the proof in [11, Pr. 2.2.2,2.2.3]). Note that feasible complete routes (9) satisfy this definition, i.e., $\mathbb{A} = (\mathbb{I} - \mathrm{bi})[\overline{1,N}]$ [11, Th. 2.2.1].

The complete problem is to visit the set of megalopolises $\overline{1,N}$, starting from a separate point, the base x^0. Subproblems resemble the full problem inasmuch as they involve visiting a feasible subset $K \in \mathfrak{N}$, starting from some $x \in \mathbb{X}_{\mathrm{out}}$. However, there has to be an additional restriction on the base point for subproblems: the movement from the base to an element of K we decide to visit first must be feasible with respect to precedence constraints. Once more, the mapping $\mathbb{I}$ is used to formally express this feasibility: let $x \in \mathbb{X}_{\mathrm{out}}$ belong to $\mathbb{M}_i^{(\mathrm{out})}$ for some $i \in \overline{1,N} \setminus K$. Clearly, the movement from x to the element of K we will visit first is to be considered feasible if and only if the megalopolis M_i *may* occupy the first place in a partial route over $\{i\} \cup K$; thus, $x \in \mathbb{M}_i^{(\mathrm{out})}$ is said to be a *feasible base* for the task set K if and only if $K \cup \{i\}$ is a feasible set and $i \in \mathbb{I}\big(K \cup \{i\}\big)$.

We also need to define the set of partial *tracks* that agree with a given partial route. For $K \in \mathfrak{N}$, a feasible base $x \in \mathbb{X}_{\mathrm{out}}$, and $\alpha \in (\mathbb{I} - \mathrm{bi})[K]$, let

$$\mathcal{Z}(x,K,\alpha) \triangleq \left\{ (z_i)_{i=0}^{|K|} \in \mathcal{Z}_K \,\middle|\, \big(z_0 = (\{\varnothing\}, x)\big) \& \big(z_t \in \mathbb{M}_{\alpha(t)}\ \forall t \in \overline{1,|K|}\,\big) \right\}. \quad (17)$$

We can finally define the quality criterion for subproblems

$$\begin{aligned} \mathfrak{C}_K^{(\alpha)}\big[(z_i)_{i=0}^{|K|}\big] \triangleq \max_{t \in \overline{0,|K|-1}} \Big[& \mathbf{c}\Big(\mathrm{pr}_2(z_t), \mathrm{pr}_1(z_{t+1}), \big\{\alpha(s) : s \in \overline{t+1,|K|}\,\big\}\Big) + \\ & + c_{\alpha(t+1)}\Big(z_{t+1}, \big\{\alpha(s) : s \in \overline{t+1,|K|}\,\big\}\Big)\Big]. \end{aligned} \quad (18)$$

A subproblem itself is to *minimize* this criterion by choosing the right partial *route* and *track*. Minimizing, we obtain the *value* (extremum) of the subproblem,

$$v(x,K) \triangleq \min_{\alpha \in (\mathbb{I}-\mathrm{bi})[K]}\ \min_{(z_i)_{i=0}^{|K|} \in \mathcal{Z}(x,K,\alpha)} \mathfrak{C}_K^{(\alpha)}\big[(z_i)_{i=0}^{|K|}\big] \in [0,\infty[. \quad (19)$$

A partial route and track pair $\big(\alpha^*, (z_i^*)_{i=0}^{|K|}\big)$, $\alpha^* \in (\mathbb{I} - \mathrm{bi})[K], (z_i^*)_{i=0}^{|K|} \in \mathcal{Z}(x,K,\alpha^*)$, is said to be *optimal* for the problem described by the OP (x,K) if it attains its extremum: $\mathfrak{C}_K^{(\alpha^*)}\big[(z_i^*)_{i=0}^{|K|}\big] = v(x,K)$.

Let us supplement the definition of $v(x,K)$ with the trivial case $K = \varnothing$; to find out which $x \in \mathbb{X}_{\mathrm{out}}$ we can use, we may actually apply the definition of *feasible base* to the empty set. This yields the points x belonging to all megalopolises

that are not senders, $\{i \in \overline{1,N} \,|\, \forall z \in \mathbb{K}\ i \neq \mathrm{pr}_1(z)\}$, and thus qualified to terminate a feasible (full) route. We need this trivial case to prime the recurrence procedure for of V; set

$$v(x, \varnothing) \triangleq 0\ \forall x \in \mathbb{X}_{\mathrm{out}}.$$

This initial condition serves the needs of the *open* TSP-like problems (see [15]). To accommodate for the more oft-cited *closed* TSP setting, where it is mandated to return to the starting point after walking through $\overline{1,N}$, it is only necessary to replace zero costs with the costs of going from x to the starting point. It is equally easy to introduce a more general *terminal cost* function reminiscent of that in the optimal control theory; it is explored in [11].

At long last, we are prepared to state the Bellman equation: for $K \in \mathfrak{N}$ and a feasible base $x \in \mathbb{X}_{\mathrm{out}}$,

$$v(x,K) = \min_{j \in \mathbb{I}(K)} \min_{z \in \mathbb{M}_j} \max\Big\{ \mathbf{c}\big(x, \mathrm{pr}_1(z), K\big) + c_j\big(z, K\big);\, v\big(\,\mathrm{pr}_2(z), K \setminus \{j\}\,\big)\Big\}. \tag{BF}$$

For the proof, refer to [13]; it is not too complex yet rather laborious, in no small part due to the generalized clustered character of the problem.

4 Exact dynamic programming

In this section we describe the exact DP solution of (BGTSP-PC), on which the heuristic is based. Recall that earlier we have redefined the set $\mathfrak{N}$ to be the set of all *feasible* task sets. Yet, we find it more appealing to use a different symbol in this section: let $\mathcal{G} \triangleq \big\{K \in \mathcal{P}'\big(\overline{1,N}\big) \,\big|\, \forall z \in \mathbb{K}\ (\mathrm{pr}_1(z) \in K) \Rightarrow (\mathrm{pr}_2(z) \in K)\big\}$ be the set of all feasible task sets; note the exclusion of the empty task set. It *is* regarded as "feasible", but it is more straightforward to treat it separately. We now proceed to describe the procedure of generating and traversing the set of all feasible states, where to *traverse* a state means to obtain its value (19) through (BF).

First of all, partition the feasible task sets according to their cardinality,

$$\mathcal{G}_s \triangleq \Big\{K \in \mathcal{G} \;\Big|\; s = |K|\Big\}\ \forall s \in \overline{1,N}.$$

Note the boundary elements of this partition: the last one, $\mathcal{G}_N = \big\{\overline{1,N}\big\}$ is the singleton reflecting the *complete* task set. The first element $\mathcal{G}_1$ is the set of all "nonsenders", which are eligible to terminate a feasible route; clearly, no "sender" megalopolis is eligible to do so. The remaining elements of the partition are defined in a recurrent way,

$$\mathcal{G}_{s-1} = \Big\{K \setminus \{t\} \,:\, K \in \mathcal{G}_s, t \in \mathbb{I}(K)\Big\}\ \forall s \in \overline{2,N}. \tag{20}$$

For the proof of validity of this procedure, refer to [11, Prp. 4.9.1].

Let us now describe the procedure that constructs the states based on feasible task sets. Consider the *feasible expansion* of a feasible set,

$$\mathcal{J}(K) \triangleq \big\{j \in \overline{1,N} \setminus K \,\big|\, \{j\} \cup K \in \mathcal{G}_{s+1}\big\} \in \mathcal{P}'(\overline{1,N} \setminus K), \tag{21}$$

the set of megalopolises the addition of which to K yields a *feasible* set. In view of (20), this definition links well with that of *feasible base* introduced in Section 3, namely, feasible bases for K are exactly the cities that belong to the megalopolises that form the feasible expansion of K. Let us collect all feasible bases for a task set K into the set[5]

$$\mathcal{M}[K] \triangleq \bigcup_{j \in \mathcal{J}(K)} \mathbb{M}_j^{(\mathrm{out})},$$

and construct the appropriate states,

$$\mathbb{D}[K] \triangleq \big\{(x, K) : x \in \mathcal{M}[K]\big\}.$$

Collecting $\mathbb{D}[K]$ for all feasible K of a certain cardinality, we obtain the *layers* of state space,

$$D_s \triangleq \bigcup_{K \in \mathcal{G}_s} \mathbb{D}_s[K] \in \mathcal{P}'(\mathbb{X}_{\mathrm{out}} \times \mathcal{G}_s)\ \forall s \in \overline{1, N-1};$$

let us supplement this definition with the two boundary cases $D_0 \triangleq \big\{(x, \{\varnothing\}) : x \in \mathbb{M}_j^{(\mathrm{out})}, j \in \mathcal{G}_1\big\}$ and $D_N \triangleq \big\{(x^0, \overline{1, N})\big\}$.

From the feasible *task sets*, the state space layers inherit a connection between the elements of neighbouring cardinality:

$$\big(y, K \setminus \{k\}\big) \in D_{s-1}\ \forall s \in \overline{1, N}\ \forall K \in \mathcal{G}_s\ \forall k \in \mathbb{I}(K)\ \forall y \in \mathbb{M}_k^{(\mathrm{out})}. \tag{22}$$

This connection substantiates a natural assumption that to calculate (BF) for states from D_s, it is necessary to know the values $v(\cdot, \cdot)$ (19) for all of the states from D_{s-1}. Note that it is actually *not* the *only* option, for example, in [45], a kind of "depth-first" state generation and traversal was implemented for a precedence constrained scheduling problem.

Technically, we consider the restrictions of $v(\cdot, \cdot)$ (19) onto the state space layers, i.e., restrictions onto subproblems with fixed task set cardinality,

$$\begin{aligned} &\forall s \in \overline{0, N}\ v_s \in \mathcal{R}_+[D_s]; \\ &\qquad v_s(x, K) \triangleq v(x, K)\ \forall (x, K) \in D_s. \end{aligned}$$

Then, we say that for all $s \in \overline{1, N}$, the function $v_s \in \mathcal{R}_+[D_s]$ is obtained from the function $v_{s-1} \in \mathcal{R}_+[D_{s-1}]$ through "stratified" (BF),

$$\begin{aligned} &v_s(x, K) = \min_{j \in \mathbb{I}(K)} \min_{z \in \mathbb{M}_j} \max\Big\{\mathbf{c}\big(x, \mathrm{pr}_1(z), K\big) + c_j\big(z, K\big);\ v_{s-1}\big(\mathrm{pr}_2(z), K \setminus \{j\}\big)\Big\}; \\ &v_0\big(\hat{x}, \{\varnothing\}\big) = 0,\ \big(\hat{x}, \{\varnothing\}\big) \in D_0; \qquad (\mathrm{BF}_s) \\ &\qquad\qquad v_0 \longrightarrow v_1 \longrightarrow \ldots \longrightarrow v_N = V. \end{aligned}$$

[5] In an ordinary, non-generalized TSP, there is no need for a separate set $\mathcal{M}[K]$, or rather, this set would match the feasible expansion of K.

Note that feasible sets and, therefore, feasible states, are generated *top-to-bottom* (20), whereas the costs are calculated *bottom-up*; such implementation of (BF_s) would first generate all the states and only after that start to calculate their values. Our implementation did the generation and computation at the same time, which was made possible by the *bottom-up* generation procedure stemming from the following alternative definition of *feasible expansion*

$$\mathcal{J}(K) = \left\{ j \in \overline{1,N} \setminus K \;\middle|\; \forall z \in \mathbb{K} \left[\big(j \neq \mathrm{pr}_1(z)\big) \vee \big((\mathrm{pr}_1(z) = j) \Rightarrow (\mathrm{pr}_2(z) \in K)\big) \right] \right\}. \tag{23}$$

It can be proved that both implementations generate the same set of feasible states. More sophisticated procedures for generation of feasible sets are known in the literature that were never, to the best of author's knowledge, applied to precedence constrained discrete optimization problems. In the perspective of partial order theory, a feasible set is nothing else than an *order ideal*, see the list of algorithms for generating them in [9, Apx. 2.2].

After the value $V = v_N\big(x^0, \overline{1,N}\big)$ of the complete problem is found through backwards traversal of (BF_s), we need to obtain the actual solution of the problem, the optimal route and track. Starting with v_N, at each step from v_s to v_{s-1}, append $\mathbb{j}_s \in \mathbb{I}(K)$ and $\mathbb{z}_s \in \mathbb{M}_{\mathbb{j}_s}$ that yield the corresponding extremum in (BF_s) to the end of optimal route and track. There may be multiple optimal routes and tracks, more so for a bottleneck problem, and it is possible to obtain all of them through this procedure; however, we were only concerned with finding *some* optimal solution.

5 Parallel implementation of exact DP

The algorithm specified in the previous section was implemented in `C++` with the work sharing done through the `OpenMP` shared memory multiprocessing API. A similar parallel implementation was reported in [21] for the additive GTSP-PC without sequence dependence; a similar non-`OpenMP` based on `C#` threads was reported in [27] for the same problem. A different parallelization strategy for GTSP-PC was reported in [12]. The *divide-and-conquer* approach of [33] may be applied to GTSP-PC in a way that will not invalidate precedence constraints [44]; the author is not aware of applications of the divide-and-conquer strategy to DP for precedence-constrained problems.

Let us first recall the general structure of the algorithm:

1. Prime the *feasible state* generation with the layer D_0 of states with empty task sets. Generate the *feasible states* $D_1, \ldots, D_N$ through procedure (23).
2. Prime the calculation of V, the value of the complete problem, with the trivial values $v_0(\cdot,\cdot) = 0$. Calculate V through recurrence equation (BF_s).
3. Recover *an* optimal solution (a route and a track that agrees with it) based on the optimal values of the Bellman function $v(\cdot,\cdot)$, starting with the value of the complete problem, V.

Recall that, for each $k \in \overline{1, N}$, to generate the next state space layer D_k with the aid of *bottom-up* procedure (23), we only need to know the previous layer D_{k-1}; the same is true for the values of the states as specified in (BF_s). Thus, steps 1 and 2 can in fact be conducted *almost* simultaneously: for all $i \in \overline{0, N-1}$, from $\mathcal{G}_i$ we generate both the next set of feasible task sets $\mathcal{G}_{i+1}$ and the "current" state space layer D_i with the aid of $\mathcal{J}(\cdot)$ (23), supplemented by $\mathcal{M}[\cdot]$ and $\mathbb{D}[\cdot]$ in the latter case. As soon as a state in layer D_i is generated, it can be priced because the values for the previous layer D_{i-1} are already known. Figures 1 and 2 exhibit this process for two neighbouring iterations; the circled items have to be accessible for the process to go on; the single arrows denote *generation* and the double arrows denote *pricing*.

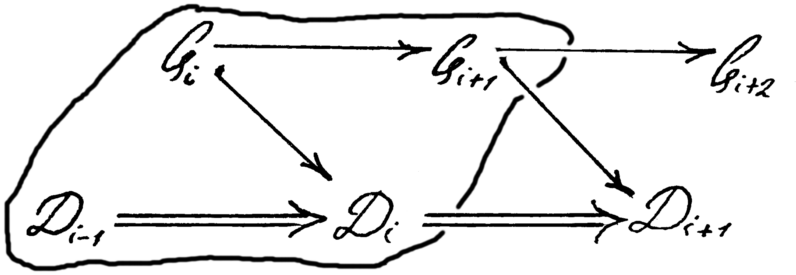

Fig. 1. Step i: Generate $\mathcal{G}_{i+1}$ and D_i from $\mathcal{G}_i$. Price D_i with the aid of values of D_{i-1}.

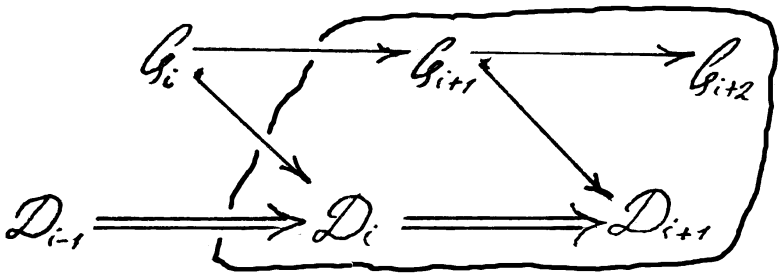

Fig. 2. Step $i+1$: Generate $\mathcal{G}_{i+2}$ and D_{i+1} from $\mathcal{G}_{i+1}$. Price D_{i+1} with the aid of values of D_i.

The parallel work sharing through `OpenMP` happens in the treatment of the current set of feasible task sets $\mathcal{G}_i$. Each of the feasible task sets can be processed independently, and since the states corresponding to different task sets never coincide, no race conditions occur in generation and pricing of the feasible states set D_i. On the contrary, it is possible that some $K' \in \mathcal{G}_{i+1}$ could be generated from both $K_1 \in \mathcal{G}_i$ and $K_2 \in \mathcal{G}_i$, $K_1 \neq K_2$, a trivial example of which is the final $\mathcal{G}_N$; see the paragraph below for an exception to this rule. To avoid race conditions, an `omp critical` directive had to be placed around the operation of

writing to $\mathcal{G}_{i+1}$ to prevent another thread from writing there at the same time. Better algorithms for generating the feasible task sets would eliminate this race condition altogether since they never generate a new feasible task set more than once, for more details on those algorithms, see the list [9, Apx. 2.2] and the corresponding papers. The one we used was related to the algorithm from [33] and hence at most quadratic in the total number of feasible task sets.

Speaking pedantically, the precedence constraints $\mathbb{K}$ may in fact mandate a *total* order on $\overline{1,N}$. In that case the *route* is actually fixed from the start and only an optimal *track* is to be determined; a Generalized TSP under such conditions is known as *Serdyukov* TSP, and there is also *Ordered Cluster* TSP [23, p. 26], with the usual difference between Generalized and Clustered TSP as regards the cities to be visited in each megalopolis. In this case, it is futile to do a parallel processing of elements of $\mathcal{G}_i$ since each of them is a singleton. Still, the corresponding state space layers D_i do not degenerate into singletons and it is possible to share the work by pricing the newly generated elements of each D_i in parallel.

Data structures. Our principal data structure was *nested hash table* implemented with the aid of `std::unordered_map` (hence `C++11`). Its type can be specified as `std::vector<std::unordered_map<uint32_t, std::unordered_map<uint16_t, float>>>`, where `uintXX_t` stands for `XX`-bit unsigned integer type. Semantically, to access the cost of walking through $K \in \mathcal{G} \cup \{\varnothing\}$ starting from $x \in \mathcal{M}[K]$, i.e., $v(x, K)$, we had to call the subscript operator *three* times: `[|K|][K][x]`. The task sets were coded in the standard bit-masking way; our implementation provided for up to 31 megalopolises; zero was reserved for the base. The cities were numbered such that the numbers for all cities in each megalopolis were consecutive. A *nested* structure let us avoid the unnecessary repetition of `32`-bit task set labels at every state that corresponds to the same task set, of which there is a non-negligible number in *generalized* problems; for non-generalized problems, the flat structure as specified in, for example, [36], may be justified. An additional benefit of nested structure was the ability to generate the next (in the sense of task set cardinality) feasible states layer by only going through the corresponding set of feasible task sets, without resorting to examining the whole list of present feasible states, which is considerably larger, more so in a generalized problem such as ours.

We adopted *hash table* as a base data structure because of its amortized constant-time search and the lack of a need to remove elements once installed; since both keys we used were unsigned integer numbers, we did not have to invent our own hash function. Our first experiments were with `std::map`, which offered logarithmic search time, but the switch to `std::unordered_map` made the computation time decrease by a factor of 4 (we admit to only comparing their performance on a single test problem), at which point we settled with the hash table. On the flip side, the use of a hash table did little to help decrease the memory footprint, this perennial problem of dynamical programming for TSPs; it also became rather difficult to predict that footprint. We did not do rigorous memory profiling, but we can still say that 4GB of RAM were enough for the

solution of the 27–25–25 problem, and 30–25–25 (see the problem descriptions in Sect. 6) took more than 15GB and less than 40GB. The use of hash table also mandated the implementation of `OpenMP tasks` work-sharing construct, which only became available in the third version of the shared memory API, with a single task being the processing of a single feasible task set, i.e, the generation and pricing of its corresponding states and the generation of "successor" feasible task sets. The pseudocode of `OpenMP tasks` implementation is listed below, where the blocks are defined by indentation instead of braces as would befit `C++`.

For each $s \in \overline{1, N-1}$
`#pragma omp parallel default (shared)`
`#pragma omp single nowait`
For each $K \in \mathcal{G}_s$
`#pragma omp task untied firstprivate`(K)
Generate $\mathcal{J}(K)$
For each $j \in \mathcal{J}(K)$
`#pragma omp critical (expand)`
`foo[|`K`|+1].add(`$K \cup \{j\}$`)`
For each $x \in \mathcal{M}[K]$
`Compute` $v_s(x, K)$
`#pragma omp critical (costwrite)`
`foo[|`K`|][`K`][`x`]:=`$v_s(x, K)$

5.1 Experiment

For problem descriptions, refer to Subsect. 6.1. Here and below, the absolute computation time and the scaling factor (serial run time divided by the run time considered) are expressed as "MM:SS —— Ratio". The times specified include input-output operations. The following results were reported in [42]:

Table 1. PC-BGTSP OpenMP Speedup I

Problem	srl	par-4	par-8
27–10–25–NO	04:58 ——1	01:25 ——3.506	00:44 ——6.773
27–20–25–NO	38:45 ——1	10:34 ——3.667	05:18 ——7.311
27–25–25–NO	73:32 ——1	20:27 ——3.596	10:13 ——7.197

They were all obtained on the Uran supercomputer (for details, see `http://parallel.uran.ru/node/6` [in Russian]) at IMM UrB RAS. The Uran supercomputer ran 64-bit `Scientific Linux 6.4`; the compiler used was `GCC 4.4.7`, optimization level `-O2`. Evidently, the speed-up was near-linear, as much as it could scale for a shared-memory implementation. Eight was the maximum number of cores per node that was available at the time.

Computation times from a more recent implementation of the algorithm, which featured a streamlined code, a probable source of improvement through aiding optimizing compiler rather than trying to actually optimize the code, are listed below:

Table 2. PC-BGTSP OpenMP Speedup II

Problem	srl	par-2	par-3	par-4
27–10–25–NO	02:57 ——1	01:27 ——2.034	01:00 ——2.95	00:46 —— 3.848
27–20–25–NO	22:34 ——1	11:28 ——1.968	08:08 ——2.775	05:53 —— 3.836
27–25–25–NO	42:56 ——1	22:23 ——1.918	14:48 ——2.901	11:23 —— 3.772

These computations were carried out on the author's PC (`Intel Core i3-3450`) in 64-bit `Windows 7` environment. The compiler used was `Intel C++ 15 Update 1` since `Microsoft Visual C++` would not support the required `OpenMP 3.0` API; the compilation parameters were copied over from the default "Release" configuration, with the obvious addition of a flag that allows `OpenMP` code generation. The superlinear speedup as seen on the two-core calculation for 27–10–25–NO is most probably caused by `OpenMP tasks` overhead.

As evident from the two tables above, the proposed shared-memory parallel implementation scheme that works through `OpenMP tasks` provides a reasonable speedup for the problems considered, with the main barrier being the number of cores on a single node that could work through `OpenMP`. It seems to be possible to make a similar *message-passing* implementation, however, the latter seems to be sensible only for the problems of greater magnitude than those specified above, either through a greater number of megalopolises or cities therein, or less strict precedence constraints, as it would certainly impose a greater overhead. However, a rudimentary message-passing implementation might be used to divide the work between two nodes with the aim of using only half as much memory on each node as required for a single-node (shared memory) implementation, see [33, 44].

6 Restricted DP heuristic. Experiment

The heuristic we implement was proposed for Time-Dependent TSP in [36], and then reviewed and implemented as 'a framework for solving realistic Vehicle Routing Problems' [22]. It consists of retaining only H, $H \in \mathbb{N}$, *best*, as measured by their values, states at each state space layer; we call H the *depth parameter* or simply *depth*. Naturally, if H surpasses the cardinality of the most populous state space layer, this heuristic turns into exact dynamic programming. On the other hand, fixing $H = 1$ yields the well-known nearest neighbour heuristic.

In contrast with the exact algorithm, the base data structure chosen for the heuristic was `std::map`. Our intention was to implement a heuristic that

is reasonably fast and has a minimum memory footprint. We did not calculate the Pareto optimum, but we figured that the linear search time associated with the use of a common array for states will not meet our definition of "fast" and thereby took the compromise data structure. We used (`std::deque`) for keeping a "sorted list" of states when searching for the H best at each layer, but linear search time was not an issue here since removal only happened from the end (i.e., the worst state would go when a better one was obtained), and the addition of a state mandated re-sorting the container (done through `std::sort`) anyway. Another sacrifice made to diminish the memory footprint was the lack of buffering when generating and pricing the new states, i.e., they were examined one at a time.

The heuristic is based on a *restricted* recurrence relation reminiscent of the exact Bellman function in (BF_s). All "restricted" items in the expressions below and elsewhere are denoted by placing a tilde $\widetilde{\ }$ above the respective notation used for the "exact" items.

$$\widetilde{v}_s(x,K) = \min_{j\in\widetilde{\mathbb{I}}(K)}\min_{z\in\widetilde{\mathbb{M}}_j}\max\Big\{\mathbf{c}\big(x,\mathrm{pr}_1(z),K\big)+c_j\big(z,K\big);\,\widetilde{v}_{s-1}\big(\mathrm{pr}_2(z),K\setminus\{j\}\big)\Big\};$$

$$\widetilde{v}_0\big(\hat{x},\{\varnothing\}\big)=0,\ \big(\hat{x},\{\varnothing\}\big)\in D_0; \qquad (\widetilde{BF}_s)$$

$$\widetilde{v}_0\longrightarrow\widetilde{v}_1\longrightarrow\ldots\longrightarrow\widetilde{v}_N=\widetilde{V}.$$

The main difference between (BF_s) and ($\widetilde{BF}_s$) lies in a restriction of $\mathbb{I}(\cdot)$ and $\mathbb{M}_j$: whereas in the exact procedure, for each layer $D_1,\ldots,D_N$, the existence of $v_{s-1}\big(\mathrm{pr}_2(z),K\setminus\{j\}\big)$ is guaranteed for all $j\in\mathbb{I}(K)$ and $z\in\mathbb{M}_j$, the minimization over which yields $v_s(x,K)$, it is not always the case for a restricted procedure. It is entirely possible that some state $\big(\bar{l},K\setminus\{\bar{l}\}\big)\in D_{s-1}$ *did not make it* into the H best that formed $\widetilde{D}_{s-1}\subseteq D_{s-1}$; its (heuristic) value $\widetilde{v}\big(\bar{l},K\setminus\{\bar{l}\}\big)$ was not retained and could not be used in the computation of $\widetilde{v}_s(x,K)$. Hence the need for $\widetilde{\mathbb{I}}(K)\subseteq\mathbb{I}(K)$ and $\widetilde{\mathbb{M}}_j\subseteq\mathbb{M}_j$ that retain only the elements $j\in\mathbb{I}(K)$ and $(z_{\mathrm{in}},z_{\mathrm{out}})\in\mathbb{M}_j$ for which the states $\big(z_{\mathrm{out}},K\setminus\{j\}\big)$ were among the H best that formed $\widetilde{D}_{s-1}$. Since in the restricted case the domain over which the minimization is conducted becomes smaller, we obviously have $v_s(x,K)\le\widetilde{v}_s(x,K)\ \forall s\in\overline{1,N}$ for all (x,K) for which $\widetilde{v}_s$ was calculated, thus, ($\widetilde{BF}_s$) provides an *upper* bound $\widetilde{V}$ for the value V. The outline of the algorithm is as follows:

1. Prime the algorithm with $\widetilde{\mathcal{G}}_0=\mathcal{G}_0$, $\widetilde{D}_0=D_0$, and $\widetilde{cG}_1=\mathcal{G}_1$. The depth requirement is not imposed on $\widetilde{D}_0$ because all the states have the value of zero.
2. For each $l\in\overline{1,N-1}$
   - For each $K\in\widetilde{\mathcal{G}}_l$
     Generate its *feasible expansion* $\mathcal{J}(K)$
     For each $j\in\mathcal{J}(K)$
     * For every $x_{\mathrm{out}}\in\mathbb{M}_j^{(\mathrm{out})}$
       (a) Calculate $\widetilde{v}(x_{\mathrm{out}},K)$

(b) If $|\widetilde{D}_l| = H$ and $\exists (y, K') \in \widetilde{D}_l : \widetilde{v}(y, K') > \widetilde{v}(x_{\text{out}}, K)$, remove (y, K') from $\widetilde{D}_l$ and add (x, K) to $\widetilde{D}_l$.

– For each $(x, K) \in \widetilde{D}_l$
  Let $i \in \mathcal{J}(K)$ be such that $x \in \mathbb{M}_i^{(\text{out})}$. Add $K \cup \{i\}$ to $\widetilde{\mathcal{G}}_{l+1}$.

3. Calculate $\widetilde{v}(x^0, \overline{1, N}) = \widetilde{V}$.
4. Recover a route and track that conform to $\widetilde{V}$ (when substituted into (13), yield at most $\widetilde{V}$).

The recovery procedure that obtains a route and track yielding at most $\widetilde{V}$ when substituted into quality criterion (13) from the values of $\widetilde{v}(\cdot, \cdot)$ differs from the same procedure for the exact DP as much as (BF_s) differs from $(\widetilde{BF}_s)$; we will omit the details. The only interesting difference is the fact that a heuristic solution could possibly perform better than the corresponding heuristic value $\widetilde{V}$ because not all theoretically available information is actually used to determine $\widetilde{V}$ (some is lost as the states are dropped). However, our model problems did not exhibit this behaviour.

For an upper estimate of complexity of the algorithm, let us fix some more constants. Recall that we have N megalopolises plus the base. Let each megalopolis have at most m cities. Let b denote the constant time required to calculate the summary cost of exterior movement $\mathbf{c}\big(x, \text{pr}_1(z)\big)$ and interior job $c_i\big(\text{pr}_1(z), \text{pr}_2(z)\big)$; the latter assumption is correct if all possible interior jobs are either simple or pre-calculated. Assume the time to compute the maximum of two values is also included in b.

Let us now consider, how long does it take to compute the heuristic value for a state (x, K). The generation of a feasible state K takes at most N^2 operations (see [9, Apx. 2.2]); then, we have to actually compute $\widetilde{v}(x, K)$, to which end we need to consider all $i \in \widetilde{\mathbb{I}}(K)$, of which there are never more than N, and at most m^2 pairs $(z_{\text{in}}, z_{\text{out}})$ which form the set $\mathbb{M}_i$; for each such pair, it takes b to calculate

$$\max\Big\{\mathbf{c}(x, z_{\text{in}}) + c_i(z_{\text{in}}, z_{\text{out}}); v_{|K|-1}\big(z_{\text{out}}, K \setminus \{i\}\big)\Big\}.$$

Thus, to calculate $(\widetilde{BF}_s)$ for a state (x, K), it takes at most $N^2 \cdot N \cdot m^2 \cdot b = N^3 m^2 b$.

There are $N + 1$ layers, each having at most H states, with the exception of D_0, which is not constrained by H, but still has at most N states. The last layer D_N always has a single state $\big(x^0, \overline{1, N}\big)$. Thus, the computation of $\widetilde{V}$ takes at most

$$(N + (N-2)H + 1)N^3 m^2 b = \mathcal{O}(N^4 H m^2 b). \tag{24}$$

The search for a conforming solution necessitates the examination of, in the worst case, all $(N+(N-2)H+1)$ states. The examination consists of checking if it was indeed the given state (x, K) that led to $v\big(y, K \cup \{j\}\big)$; thus, it is the same as actually calculating $v\big(y, K \cup \{j\}\big)$ without the need to generate it, hence the cost $(N+(N-2)H+1)Nm^2b$, which does not change the $\mathcal{O}$-value in (24). This is a rather generous estimate, since a feasible task set K is actually only generated

once for all states that contain it, and not all of m^2 pairs $(z_{\text{in}}, z_{\text{out}}) \in \mathbb{M}_i$ have to be examined each time since the state that has z_{out} might have been left out of H best. The same can be said with respect to $\widetilde{\mathbb{I}}(K)$, the cardinality of which is actually at most K, with this bound being true only for K that are antichains.

6.1 Experiment

The model problems were borrowed from our previous research in exact solutions of (BGTSP-PC), namely, the sequence-dependent case [13] and the previously considered "sequence-independent" cases [41, 42]. The model problems were considered on a subset of Euclidean space $X = [0, 1024] \times [0, 768] \subset \mathbb{R} \times \mathbb{R}$ for $N_1 = 30$ and $N_2 = 27$ megalopolises with $|\mathbb{K}| = 25$ precedence constraints and 25 cities in each megalopolis; each city could serve as both exit point and entry point.-

The cost of *exterior movement* was specified as *Euclidean distance* in X; for the sequence dependent-case, it was multiplied by a trivially sequence-dependent coefficient $a(|K|) = 1 + \frac{N-|K|}{N}$. The cost of *interior jobs* was the *Manhattan norm* $||\cdot||$ of movement from the "entry point" into the megalopolis to the "exit point" through its *center* (for two plane vectors $x = (x_1, x_2), y = (y_1, y_2)$, the Manhattan norm is $||x - y|| = |x_1 - y_1| + |x_2 - y_2|$); this type of interior jobs is closer to the classical Generalized TSP than to Clustered TSP.

Megalopolises were modeled as equal radius disks, and the cities were placed on the circumference with equal angular distances between them (which, obviously, depended on the number of cities in the megalopolis); megalopolises were distributed randomly. Below, dimensions of the problems are encoded in the form "X–Y–Z–W", where X is the number of megalopolises, Y is the number of cities per megalopolis, Z is the number of precedence constraints, and W is either "SD" for the sequence dependence as specified above or "NO" for problems without sequence dependence. In the two 30–25–25–* problems, the geometry is the same (cities have the same coordinates). All data sets are available from the author on request.

Since it is the exact form of precedence constraints and not just their number that influences the complexity of solving the appropriate problem via dynamic programming [46, 47, 43], we list them below, in our preferred address pairs form: (1,10); (12,2); (2,13); (13,15); (6,16); (15,16); (18,27); (9,27); (10,9); (11,19); (20,19); (25,26); (23,22); (21,20); (24,22); (14,16); (7,10); (8,2); (1,9); (14,26); (2,27); (3,6); (3,19); (18,17); (14,25).

It is not reasonable to directly relate these dimension parameters to the top results for TSP-PC [19] since the combination of generalized nature of the problem and precedence constraints on *megalopolises*, in absence of a requirement to visit of all *cities* of a megalopolis, precludes a direct transformation into a generic (bottleneck) TSP-PC. Still, the number of cities taken into account is rather considerable for a highly constrained problem, $n_1 = 30 \cdot 25 = 750$ cities and $n_2 = 27 \cdot 25 = 675$ cities, respectively. Computation times are specified in the HH:MM:SS or MM:SS format.

All algorithms for all problems were encoded in `C++11`; exact algorithms were parallelized with the aid of the shared memory multiprocessing API `OpenMP 3.0`. Since a heuristic is meant to be simple to compute, we did not make a parallel implementation, yet, it is possible make such an implementation with the same means. The exact programs were run on the Uran supercomputer (for details, see `http://parallel.uran.ru/node/6` [in Russian]) at IMM UrB RAS, and the heuristic was run on the author's PC (`Intel Core-i4-3450`, 16 GB RAM). The Uran supercomputer ran 64-bit `Scientific Linux 6.4`; the compiler used was `GCC 4.4.7`, optimization level `-O2`. The author's PC ran 64-bit `Windows 7`, the compiler used was `Microsoft Visual C++ 2013` with default optimization options ("Release" configuration).

6.2 27–25–25–NO

An exact solution was reported at the conference [42]. The value of the problem was $V = 341.962$. It took the Uran supercomputer 01:13:32 on a single core, 00:20:27 on 4 cores, and 00:10:13 on 8 cores to arrive at this conclusion; each core ran a single thread. The computation times relate as 7.197:2.002:1. The single-core to four-core relation is 3.596:1.

6.3 30–25–25–NO

An exact solution was reported at the conference [42]. The value of the problem was $V = 316.68$. It took the Uran supercomputer 01:42:48 to arrive at this conclusion on 8 cores.

6.4 30–25–25–SD

An exact solution of this problem was reported in [13]; the value of the problem was $V = 376.63$, and the computation took the Uran supercomputer 01:46:34 on 12 cores.

Table 3. Restricted DP for 27–25–25–NO

H	1	10	100	150	250	1000	2500	5000	10000	20000
Time	00:02	00:02	00:02	00:03	00:03	00:10	00:33	01:33	04:52	15:37
$\widetilde{V}$	1192.12	1182.04	602.763	836.27	839.793	613.632	613.632	534.167	419.913	386.139
$\widetilde{V}/V$	3.49	3.46	1.76	2.45	2.46	1.79	1.79	1.56	1.23	1.13

Table 4. Restricted DP for 30–25–25–NO

H	1	10	100	150	250	1000	2500	5000	10000	20000
Time	00:03	00:03	00:03	00:04	00:04	00:12	00:44	01:58	06:14	20:17
$\widetilde{V}$	1072.81	966.038	784.037	798.953	606.083	497.729	391.787	391.787	316.68	316.68
$\widetilde{V}/V$	3.39	3.05	2.48	2.52	1.91	1.57	1.24	1.24	1	1

Table 5. Restricted DP for 30–25–25–SD

H	1	10	100	150	250	1000	2500	5000	10000	20000
Time	00:03	00:02	00:03	00:03	00:04	00:13	00:44	02:01	06:00	20:41
$\widetilde{V}$	972.557	966.038	768.836	784.037	579.734	634.734	487.558	376.63	376.63	376.63
$\widetilde{V}/V$	2.58	2.56	2.04	2.08	1.54	1.69	1.29	1	1	1

6.5 Conclusion

For all of the problems considered, the proposed heuristic found near-optimal solutions in a reasonable amount of time. The memory footprint of the heuristic was quite small as compared to that of the exact procedure: at most 100MB were necessary for 20000-deep solution of 30–25–25–NO, which is quite low as far as DP is concerned. Two problems were solved to optimality (30–25–25–NO and 30–25–25–SD), and one (27–25–25–NO) was solved to within 13% of optimal. The run time did not exceed 30:00.

Our intention for implementing a DP-compliant heuristic was the possible use of the latter in a Morin–Marsten branch-and-bound strategy for dynamic programming [39] to overcome the memory limitations and possibly improve the computation times. The results of experiments with the heuristic can be considered proof-of-concept: for small depth parameters (up to 250), the computation times stayed reasonably small while the result exhibited a marked improvement over the greedy algorithm (the $H = 1$ column) and the larger depth parameters yielded near-optimal results. Thus, a large depth parameter may yield a decent upper bound in a reasonable time. To finally implement a branch-and-bound solution, we still need a *lower bound* for the problem. We are not aware of lower bound algorithms that specifically target precedence-constrained BTSP or generalized BTSP, therefore, the search has to start at general-purpose lower bound algorithms for plain BTSP; for a most recent treatment of such, refer to [32].

It is also interesting to note how the heuristic tends to stick to a locally best solution as the depth increases beyond a certain number ($H = 150, H = 250$ in 27–25–25–NO, $H = 150$ in 30–25–25–NO and 30–25–25–SD).

Acknowledgement. This work was supported by the Russian Foundation for Basic Research (project no. 13-08-00643). The author would also like to express his gratitude to the anonymous reviewers who pointed out the important shortcomings of this paper and made it possible to rectify them in the final version.

References

1. Aho, A.V., Garey, M.R., Ullman, J.D.: The transitive reduction of a directed graph. SIAM Journal on Computing 1(2), 131–137 (1972)
2. Alatartsev, S., Stellmacher, S., Ortmeier, F.: Robotic task sequencing problem: A survey. Journal of Intelligent & Robotic Systems pp. 1–20 (2015)
3. Alkaya, A.F., Duman, E.: A new generalization of the traveling salesman problem. Appl. Comput. Math 9(2), 162–175 (2010)
4. Allahverdi, A., Gupta, J.N., Aldowaisan, T.: A review of scheduling research involving setup considerations. Omega 27(2), 219–239 (1999)
5. Applegate, D.L., Cook, W.J., Bixby, R.E., Chvátal, V.: The traveling salesman problem. Princeton Univ. Press (2006)
6. Balas, E., Fischetti, M., Pulleyblank, W.R.: The precedence-constrained asymmetric traveling salesman polytope. Mathematical programming 68(1-3), 241–265 (1995)

7. Bellman, R.: Dynamic programming treatment of the travelling salesman problem. Journal of the ACM (JACM) 9(1), 61–63 (1962)
8. Bianco, L., Mingozzi, A., Ricciardelli, S., Spadoni, M.: The traveling salesman problem with precedence constraints. In: Papers of the 19th Annual Meeting/Vorträge der 19. Jahrestagung. pp. 299–306. Springer (1992)
9. Caspard, N., Leclerc, B., Monjardet, B.: Finite ordered sets: concepts, results and uses. No. 144 in Encyclopedia of Mathematics and Its Applications, Cambridge University Press (2012)
10. Cheblokov, I.B., Chentsov, A.G.: About one route problem with interior works [in Russian]. Vestnik Udmurtskogo Universiteta. Matematika. Mekhanika. Komp'yuternye Nauki (1), 96–119 (2012)
11. Chentsov, A.G.: Extremal problems of routing and scheduling: a theoretical aproach [in Russian]. Izhevsk:¡¡Regular and Chaotic Dynamics¿¿ (2008)
12. Chentsov, A.G.: On a parallel procedure for constructing the bellman function in the generalized problem of courier with internal jobs. Autom. Remote Control 73(3), 532–546 (2012)
13. Chentsov, A.G., Salii, Y.V.: A model of "nonadditive" routing problem where the costs depend on the set of pending tasks. Vestnik YuUrGU. Ser. Mat. Model. Progr. 8(1), 24–45 (2015)
14. Chisman, J.A.: The clustered traveling salesman problem. Computers & Operations Research 2(2), 115–119 (1975)
15. Christofides, N.: The shortest hamiltonian chain of a graph. SIAM Journal on Applied Mathematics 19(4), 689–696 (1970)
16. Dieudonné, J.: Foundations of modern analysis, Pure and Applied Mathematics, vol. 10. Academic Press New York, enlarged and corrected printing edn. (1969)
17. Dolgui, A., Pashkevich, A.: Cluster-level operations planning for the out-of-position robotic arc-welding. International Journal of Production Research 44(4), 675–702 (2006)
18. Escudero, L.: An inexact algorithm for the sequential ordering problem. European Journal of Operational Research 37(2), 236–249 (1988)
19. Gouveia, L., Ruthmair, M.: Load-dependent and precedence-based models for pickup and delivery problems. Computers & Operations Research 63, 56–71 (2015)
20. Gouveia, L., Voß, S.: A classification of formulations for the (time-dependent) traveling salesman problem. European Journal of Operational Research 83(1), 69–82 (1995)
21. Grigoriev, A.M., Ivanko, E.E., Chentsov, A.G.: Dynamic programming in a generalized courier problem with inner tasks: elements of a parallel structure. Model. Anal. Inform. Sist. 18(3), 101–124 (2011)
22. Gromicho, J., van Hoorn, J.J., Kok, A., Schutten, J.: Restricted dynamic programming: a flexible framework for solving realistic vrps. Computers & operations research 39(5), 902–909 (2012)
23. Gutin, G., Punnen, A.P. (eds.): The traveling salesman problem and its variations, Combinatorial optimization, vol. 12. Springer Science & Business Media (2002)
24. Held, M., Karp, R.M.: A dynamic programming approach to sequencing problems. Journal of the Society for Industrial & Applied Mathematics 10(1), 196–210 (1962)
25. Korotayeva, L.N., Chentsov, A.G.: On a generalization of the bottleneck traveling salesman problem. Comput. Math. Math. Phys. 35(7), 853–859 (1995)
26. Korotayeva, L.N., Sesekin, A.N., Chentsov, A.G.: A modification of the dynamic programming method for the travelling-salesman problem. USSR Computational Mathematics and Mathematical Physics 29(4), 96–100 (1989)

27. Koshelev, G.N., Kosheleva, M.S.: A parallel implementation of dynamic programming in a constrained routing problem [in Russian]. In: Problems in Contemporary Mathematics and Its Applications: Proceedings of 46th International Youth School and Conference. p. 110. N.N. Krasovskii Institute of Mathematics and Mechanics, UrB RAS; B.N. Yeltsin Ural Federal University (2015)
28. Kovács, A.: Integrated task sequencing and path planning for robotic remote laser welding. International Journal of Production Research (ahead-of-print), 1–15 (2015)
29. Kubo, M., Kasugai, H.: The precedence constrained traveling salesman problem. Journal of the Operations Research Society of Japan 34(2), 152–172 (1991)
30. Laporte, G., Martín, I.R.: Locating a cycle in a transportation or a telecommunications network. Networks 50(1), 92–108 (2007)
31. Laporte, G., Osman, I.H.: Routing problems: A bibliography. Annals of Operations Research 61(1), 227–262 (1995)
32. LaRusic, J., Punnen, A.P.: The asymmetric bottleneck traveling salesman problem: Algorithms, complexity and empirical analysis. Computers & Operations Research 43, 20–35 (2014)
33. Lawler, E.L.: Efficient implementation of dynamic programming algorithms for sequencing problems. Stichting mathematisch centrum preprint (1979)
34. Lawler, E.L., Lenstra, J.K., Kan, A.R., Shmoys, D.B. (eds.): The traveling salesman problem: a guided tour of combinatorial optimization, vol. 3. Wiley New York (1985)
35. Leon, V.J., Peters, B.A.: Replanning and analysis of partial setup strategies in printed circuit board assembly systems. International Journal of Flexible Manufacturing Systems 8(4), 389–411 (1996)
36. Malandraki, C., Dial, R.B.: A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem. European Journal of Operational Research 90(1), 45–55 (1996)
37. Melamed, I.I., Sergeev, S.I., Sigal, I.K.: The traveling salesman problem. issues in theory. Automation and Remote Control 50(9), 1147–1173 (1989)
38. Minoux, M.: Programmation mathèmatique. Théorie et algorithmes. Lavoisier, 2^e edn. (2008)
39. Morin, T.L., Marsten, R.E.: Branch-and-bound strategies for dynamic programming. Operations Research 24(4), 611–627 (1976)
40. Plotinsky, Y.M.: Generalized delivery problem. Automation and Remote Control 34(6), 946–949 (1973)
41. Salii, Y.V., Chentsov, A.G.: On a bottleneck routing problem with internal tasks [in Russian]. Tambov University Reports. Series: Natural and Technical Sciences 17(3) (2012)
42. Salii, Y.V., Chentsov, A.G.: On a precedence constrained bottleneck routing problem with internal tasks [in Russian]. In: International Conference ¡¡Discrete Optimization and Operations Research¿¿ (Novosibirsk, June 24- - 28, 2013). p. 134. Sobolev Institute of Mathematics, Novosibirsk State University (2013)
43. Salii, Y.V.: On the effect of precedence constraints on computational complexity of dynamic programming method for routing problems [in Russian]. Vestnik Udmurtskogo Universiteta. Matematika. Mekhanika. Komp'yuternye Nauki (1), 76–86 (2014)
44. Salii, Y.V.: On forward and backward programming for precedence constrained routing problems and the algorithms for generation of feasible subproblems [in

Russian]. In: Bulletin of Association for Mathematical Programming. pp. 168–169. No. 13, N.N. Krasovskii Institute of Mathematics and Mechanics, UrB RAS; B.N. Yeltsin Ural Federal University (2015)
45. Schrage, L., Baker, K.R.: Dynamic programming solution of sequencing problems with precedence constraints. Operations research 26(3), 444–449 (1978)
46. Steiner, G.: On the complexity of dynamic programming for sequencing problems with precedence constraints. Annals of Operations Research 26(1), 103–123 (1990)
47. Steiner, G.: On estimating the number of order ideals in partial orders, with some applications. Journal of statistical planning and inference 34(2), 281–290 (1993)